Tevatron Beam Position Monitor Upgrade Software Design

DRAFT DRAFT DRAFT

Margaret Votava, Luciano Piccoli, Dehong Zhang, Dinker Charak
Fermilab, Computing Division, CEPA

Abstract

This document contains the proposed design for the BPM/BLM upgrade data acquisition software.

Table of Figures

1 Introduction

This document describes the design chosen for the front-end data acquisition software for the Tevatron BPM upgrade. The goal is to provide clear guidelines for implementing and delivering a system that fulfills the specifications requirements according to the document #860.

Besides the requirements, other factors have to be considered for the design in order to achieve high quality software. These are:

- Maintainability: the software should be easy to maintain and make minor changes to adapt to new requirements;
- Extensibility: software should be easily extensible. The addition of new modes of operation should be a simple task involving minimal changes that do not affect existing components;
- Flexibility: configuration of the software should be easy to modify, adapting it to new and unexpected situations.
- Portability: software can be reused on another machines (e.g. Main Injector)

With these principles in mind the expected output will be:

- A generic software framework for Beam Position Monitor systems;
- A working Tevatron BPM system that is maintainable and extensible.

The next section the current Recycler front-end software is discussed. It will serve as base for the design and implementation of the Tevatron software. The next section describes the design of the system.

2 Proposed Tevatron BPM Software

The proposed Tevatron front-end data acquisition software is based on the software developed for the Recycler (see section 4.1). Many of its components can be reused on the Tevatron systems, such as timing control modules and readout procedures. Additionally, the tevatron system would benefit from the use of the backdoor services, making it possible to control and read out data bypassing the ACNET/MOOC infrastructure.

2.1 Control

Similarly to the recycler software, the tevatron BPM software will have a control task that is responsible for receiving ACNET and backdoor command for switching between modes of acquisition. The control task will have all data acquisition tasks started at initialization[1], so no additional time to create tasks will be needed while the system is running. The control task will need however to resume or suspend tasks according to the mode selected. VxWorks takes 124 us to start a task, while it takes 24 us to resume or suspend it.

Before letting the readout task run, the control task must configure the EchoTek boards and the timing hardware. On the recycler software, the configuration is done by the readout task when it is started.

The control task will be always running at a high priority. Commands are received through an input command queue. MOOC and the backdoor send events to the queue. Status from the control task is passed back through a response mechanism. Certain types of trigger also generate events that are passed along to the control task via its command queue.

2.2 Buffering

Every data acquisition task has a data buffer associated to it. This buffer can be shared among other data acquisition tasks and also with the buffer readout tasks, which handle outside data requests (from MOOC and backdoor). For controlling access and avoiding race conditions all data buffers must be protected by semaphores.

Buffers can be used as a data destination or a data source. On a trigger, a data acquisition task may request data from the hardware, or it may request data from an internal buffer. This will be handled transparently. In both cases, the destination of the read out data will be another buffer. The ability of having a buffer as a data source helps to implement slow read out buffers, which would get input data from fast read out buffers (e.g. Fast Abort Buffer vs Slow Abort Buffer (a more detailed list of buffers is given at section 3.5.2.1).

---

[1] The recycler software has only one data acquisition task running at a time. When modes are switched, the control task starts the task for that new mode.

2.3 Data Acquisition

The system will have several readout tasks. Each one will be responsible for filling at least one DA buffer (Fast Abort, Slow Abort, BLM). Every task runs within a closed loop and remains blocked until a trigger is received through the tasks input event queue.

A TCLK, BSYNC or State Device Change can generate the trigger. These triggers arrive at the crate controller via interrupts or software function calls, which in turn create a trigger entity and send it to the input queues of the readout tasks. The trigger awakes the readout task, which performs its job. It consists of reading the latest data from its data source and write it to its data destination(s).

2.4 ACNET Communication

ACNET requests will be handled via request queues. When a MOOC receives an external ACNET request (for reading or setting) it invokes a callback. The callback is part of the tevatron BPM software, and according to the request will create and send a request to the control queue or buffer readout queue. After completing the request, the BPM tasks (control task or buffer readout task) send back a status and/or data to the callback.

2.5 Buffer Readout

The user requests for reading data buffers are received via MOOC/ACNET according to the above section. The system can has a pool of buffer readout tasks that are waiting for requests coming into the buffer readout queue. The number of tasks can be configured, and one task means that all requests will be handled serially and more than one task means that requests can be processed in parallel if the buffers to be read are different. If data from the same buffer is requested, it will be handled serially because the buffers are protected by semaphores and only one task can access its contents at a time.

2.6 Debug and Diagnostics

The backdoor scheme will be used in the tevatron BPM DA software. It will communicate with the DA similarly to ACNET/MOOC. Whenever a request comes from the labview interface that is mapped to a callback that sends the request through a queue and gets the reply from the system. The DA software will look the same from the viewpoint of the backdoor system and the ACNET/MOOC system.

2.7 Software Diagram

The following picture (Figure 1) shows the proposed tasks, queues, data and command flow for a generic BPM system. The structure shown is valid for *one* crate within the system. The blue circles represent the tasks; the green boxes are the input queues for the tasks; the yellow boxes are the data sources and data destinations; and the blue boxes are external entities (triggers and MOOC/Backdoor).
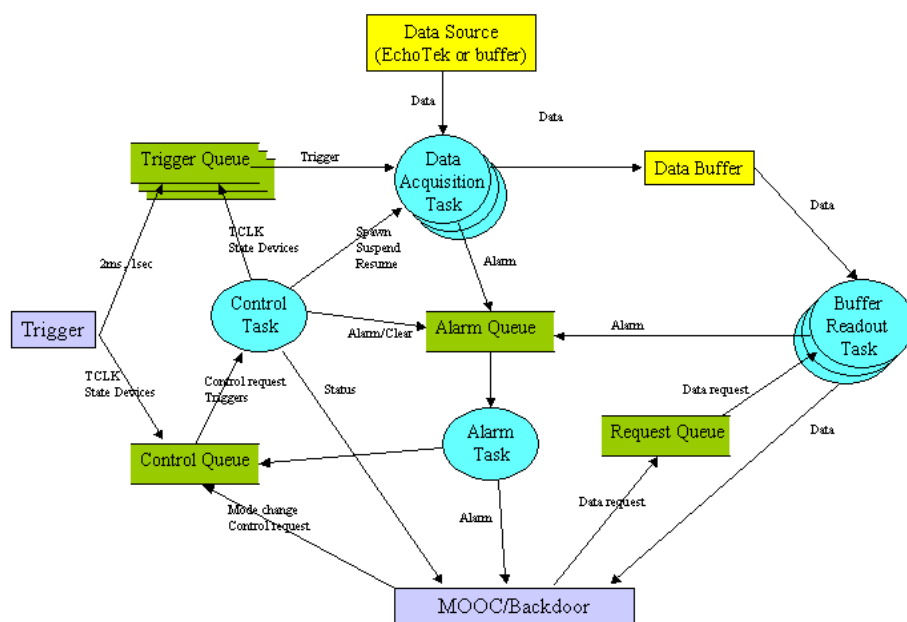
Figure 1 - Proposed tasks, queues, command and data flow

Object oriented design is used to realize the entities depicted in figure 1. The Unified Modeling Language is used to describe general use cases, classes and its relationships, control and data flows.

Figure 2 shows a specialized version for the Tevatron based on the generic BPM system (for a single crate). In the picture there are several data acquisition tasks (named *Fast Abort Task*, *Slow Abort Task* and *Turn by Turn Task*), some buffers are defined (*Fast Abort Buffer*, *Slow Abort Buffer* and *Turn by Turn Buffer*) and there is only one *Buffer Readout Task* [2].

---

[2] The picture is not complete; the system is expected to have more data acquisition tasks (e.g. *Profile Task*) and more buffers.
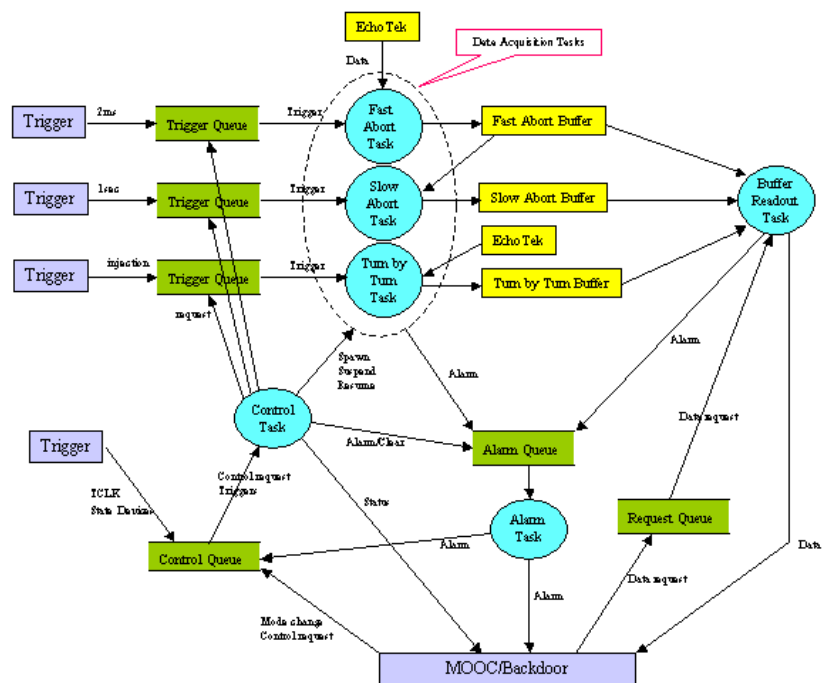
Figure 2 - Tasks for the TBPM system

3 Software Design

The remaining sections of this document describe the design of the Tevatron BPM upgrade front-end software. It takes in consideration general software quality aspects as well as aims to provide an extensible framework for future similar projects within the laboratory.

The following sections describe the use cases identified for the project, static structures and dynamic diagrams. Use cases follow the format adopted by Alistair Cockburn [Cockburn] and the notation of static and dynamic diagrams follow the UML standard [Fowler].

3.1 Use Cases

One crate in the TeV BPM DAQ system interacts with the extenal world through actions initiated by actors. The actors that interact with the system are: *User* and *Trigger*.

The User can be a control room operator, a beam physicist or another software. The User interacts with the system by *initializing* it; requesting to *change modes*; *reading out* its buffers; activating *diagnostics*. On any of these interactions there can be *alarms*, which is handled by a separate use case.

The other actor in the system, the Trigger, is any external event that is capable of changing the internal state of the system. A trigger is an input to *read out* data from acquisition boards; and input to *state device changes*.
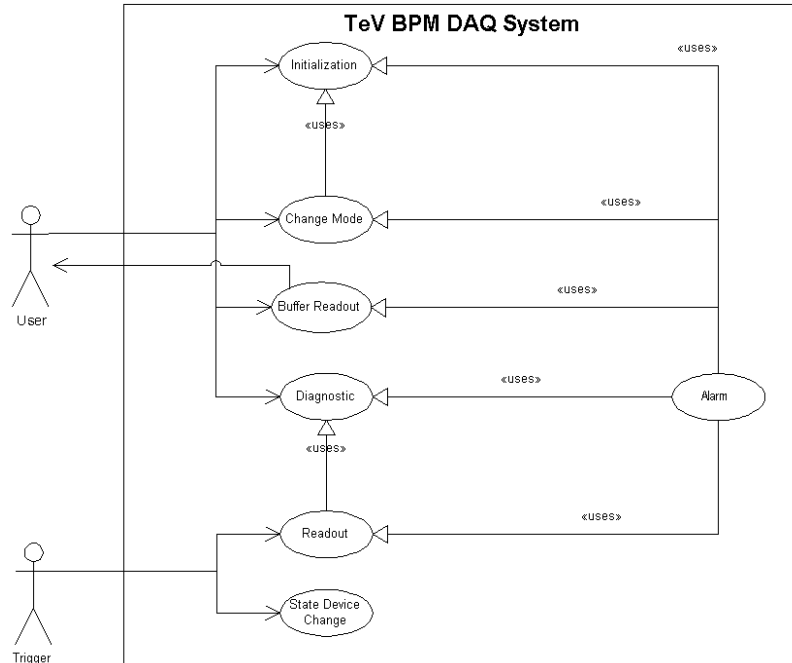
Figure 3 – Tevatron BPM front-end software use cases (need to be updated - new use case)

Figure 3 shows the use cases identified for the Tevatron BPM front-end system. Each ellipse represents one use case. The use cases are described in more detail in the following sections.

3.1.1 Initialization

3.1.1.1 Description

This use case allows the user to initialize one front-end DAQ system crate.

3.1.1.2 Basic Flow of Events

1. User ask the system (one crate) to be initialized
2. Configuration for the crate is downloaded
3. EchoTek hardware is initialized
    a. EchoTek hardware is tested (optional)
4. Timing system is initialized
    a. Timing hardware is tested (optional)
5. Read out tasks (Data acquisition) are created
6. Internal buffers are allocated
7. Buffer read out tasks are created
8. Alarm task is created
9. Alarm task is announced to the tasks in the system
10. Trigger generators are created
11. Trigger listeners are registered

12. System is enabled
13. All tasks are started
14. System is ready for use (READY state)

3.1.1.3 Alternative Flows

1. Could not download configuration (2)
   a. Report error to user
   b. Use default configuration
   c. Limit usage of the system (e.g. don't support turn-by-turn requests)

2. EchoTek card(s) did not pass test (3a)
   a. Generate internal alarm
   b. Set ALARM state
   c. Report error to user

3. Timing system did not pass test (4a)
   a. Generate internal alarm
   b. Set ALARM state
   c. Report error to user

3.1.1.4 Preconditions

None.

3.1.1.5 Postconditions

System is taking data in normal operation mode (READY state) or in a limited operational mode.

3.1.2 Mode Change

3.1.2.1 Description

This use case allows the user to request a mode change of the front-end DAQ software. There are basically two modes of operation: closed orbit and turn-by-turn. The default mode is closed orbit, and the turn-by-turn mode is enabled when the user requests it. When changing modes, the system has to reload and reprogram the data acquisition and timing hardware according to the mode specification.

3.1.2.2 Basic Flow of Events

1. User requests a mode change (from closed orbit to turn by turn)
2. An internal request for mode change is created in the system (event generator)
3. Request is posted to the control task queue
4. Request is retrieved by the control task
5. Control task checks the request
6. EchoTek boards are configured

7. Timing system is configured
8. Triggers are enabled/disabled (e.g. 2 ms closed orbit trigger)
9. Read out tasks are suspended/resumed
10. Mode has changed (CLOSED_ORBIT or TURN_BY_TURN state)

3.1.2.3 Alternative Flows

4. Mode cannot be changed (5)
   a. Return error to user
   b. Generate internal alarm

5. Requested mode change to the current mode (5)
   a. Restart mode; or
   b. Ignore request and return error.

6. Failure to change mode (6 – 9)
   a. There are conditions preventing the system to change mode

3.1.2.4 Preconditions

System is in a known operational state.

3.1.2.5 Postconditions

System has been reconfigured to run in a new mode and is acquiring or ready to acquire data.

3.1.3 Buffer Readout

3.1.3.1 Description

This use case allows users to request data from the front-end software. Data is read out from the data acquisition boards and stored in internal buffers. Data from these internal buffers are requested in this use case, and portions of it or all its contents are returned.

3.1.3.2 Basic Flow of Events

1. User requests data buffer from the system
2. An internal request is created
3. Request is posted to the buffer readout queue
4. Request is retrieved by one buffer readout task
5. The request is verified and the buffer is selected
6. Buffer is read and converted to format to be sent back to user
7. A reply with the resulting data structure returns
8. Data is sent back to the user

3.1.3.3 Alternative Flows

1. Request is not valid (5)

a.   The data requested does not exist or is out of boundaries

2. No data in the buffer (6)
     a.   Error is returned

3.1.3.4 Preconditions

Internal data buffers have data.

3.1.3.5 Postconditions

None

3.1.4 Diagnostic

3.1.4.1 Description

Use case used when user wants to get more information about the system health. Level of debug can be increased; buffers, queues and tasks are monitored more closely.

3.1.4.2 Basic Flow of Events

1. User requests system to enable diagnostics
2. An internal request is created
3. Request is posted to the control task queue
4. Request is retrieved by control task
5. Control task set the system diagnostics level

3.1.4.3 Alternative Flows

None

3.1.4.4 Preconditions

None

3.1.4.5 Postconditions

System is running at a higher diagnostics level. Performance of the system may be affected.

3.1.5 Alarm

3.1.5.1 Description

This is a use case used by other use cases in the system. It is triggered by alarm situations within the system. It is generated internally and there is no input from external actors. The alarm is handled by an alarm task, which may announce it to the external

world, depending on how critical is the situation. The system enters an alarm state which is cleared when the alarm conditions have been removed.

3.1.5.2 Basic Flow of Events

1. An internal failure is detected
2. An alarm is created
3. Alarm is posted to the alarm queue
4. Alarm task retrieves alarm from queue
5. Task evaluates the priority of the alarm
6. Task generate an external alarm, if necessary
7. Control task is informed of the alarm state
8. Control task decides the alarm is cleared
9. Alarm clear event is create
10. Alarm clear is posted to the alarm queue
11. Alarm task retrieves alarm clear from queue
12. Alarm task clear the alarm state

3.1.5.3 Alternative Flows

None

3.1.5.4 Preconditions

A failure or a potential future failure is detected.

3.1.5.5 Postconditions

System is set to an alarm state; the state can be cleared after the alarm condition is removed.

3.1.6 Readout

3.1.6.1 Description

This use case describes the actual data acquisition part of the system. The external actors involved with this use case are *triggers*. A trigger is any entity that starts the action of data acquisition. Following a trigger, the system has to perform the read out of a data source (hardware or internal buffers) and save the data to internal buffers.

3.1.6.2 Basic Flow of Events

1. A trigger is generated and received by the system
2. A trigger event is created and posted to an event queue
3. The readout task retrieves the trigger from the queue
4. Readout task performs the data acquisition
5. Data is saved in an internal buffer
6. Readout task is ready for next trigger

3.1.6.3 Alternative Flows

1. Data source is not ready to send data (4)
   a. Readout task has to wait for a defined amount of time
   b. if there is a time out an alarm is generated

3.1.6.4 Preconditions

Data acquisition hardware and timing system are configured and ready to provide data.

3.1.6.5 Postconditions

New data is saved in internal buffer and can be latter be retrieved by the user

---

3.1.7 State Device Change

3.1.7.1 Description

This use case illustrates the reaction of the system after a state device is changed. A state device can be considered an actor, more specifically a *trigger*, even though it does not trigger any data acquisition. The system has to monitor several state devices, which contain information about the accelerator status, beam type, etc. Those are important informations that are part of the metadata sent back to the user (Buffer Readout use case).

3.1.7.2 Basic Flow of Events

1. A state change is received by the system
2. A state change event is created
3. The event is posted to the control queue
4. The control task receives the event
5. Control task updates the metadata

3.1.7.3 Alternative Flows

None

3.1.7.4 Preconditions

None

3.1.7.5 Postconditions

Metadata is updated with latest state device status.

3.1.8 Configuration Change

3.1.8.1 Description

The configuration use case describes the actions taken by the user in order to change the behavior of the system. The user can specify new values for calibration, timing, filter settings, etc. During the initialization, the system receives a default configuration, and this use case represents system changes after the initialization phase.

3.1.8.2 Basic Flow of Events

1. User request a configuration change (through some mechanism)
2. A control request is created
3. Control task receives the request
4. Request is validated
5. Check if configuration can be changed
6. Change configuration

3.1.8.3 Alternative Flows

1. Request is not valid (4)
    a. Generate error
    b. Do not change configuration
    c. Generate internal alarm

2. Configuration cannot be changed (e.g. system is in turn-by-turn mode)
    a. Wait until configuration can be changed

3.1.8.4 Preconditions

System is initialized.

3.1.8.5 Postconditions

New configuration has been applied to the system.

3.2 Class Diagrams

This section describes the static structure of the system. The complete class diagram is available in the appendix section. We broken down the main diagram into pieces that handle a specific part of the system. Every piece is described bellow, each one contains a part of the full class diagram.

3.2.1 Tasks

The system has a certain number of independent processes; each one has a specific job. The tasks in the system are all subclasses of a VxWorks task wrapper (Class Task). The wrapper contains basic methods and attributes that represent a task. Figure 4 contains the task classes in the system. The upper class represents the wrapper.
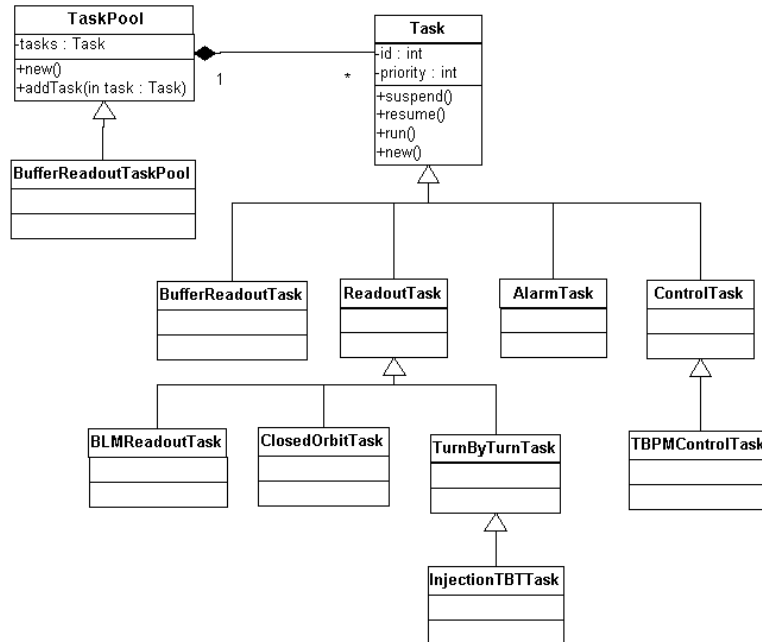

Figure 4 - Class diagram for tasks in the system

The system is overseen by a *ControlTask*, which is responsible for initializing most of the system, configure hardware (EchoTek boards and timing module), switching acquisition modes, control other tasks in the system and keep track of the overall state.

The *DataAcquisitionTask* represents the tasks that are responsible for acquiring data and storing them in internal buffers. There can be several *DataAcquisitionTask* subclasses, each one has a different acquisition method, can read data from different sources and store them in different destinations. Examples are *BLMDataAcquisitionTask*, *ClosedOrbitTask* and *TurnByTurnTask*.

BufferReadoutTask provides a standard interface to retrieve data from internal buffers. That interface can be used by MOOC and backdoor for requesting any type of data (closed orbit, turn-by-turn, display frame, etc.). The system can provide parallel access to the internal data buffers. That feature is available through the *TaskPool* class. The *BufferReadoutTaskPool* provides any number of *BufferReadoutTask*, which can handle requests in parallel.

The *AlarmTask* handles any alarms generated in the system. It is its responsibility to check the system alarm queue and decide whether to put the system in an alarm state and send an alarm to the outside world.

3.2.2 Controls

The main class in the system is the *ControlTask*. It is however controlled by the *BPM* class. The class *BPM* make a few assumptions about the system, and has common code for BPM systems in general. A more specialized class (*TBPM*) has specific implementation for the Tevatron BPM system. It contains objects of the classes *TSG* and *EchoTek*, which are the hardware present in the system VME crate. Additional hardware classes may not be shown in the diagram on Figure 5 (more are expected to come after the hardware specification document is completed).
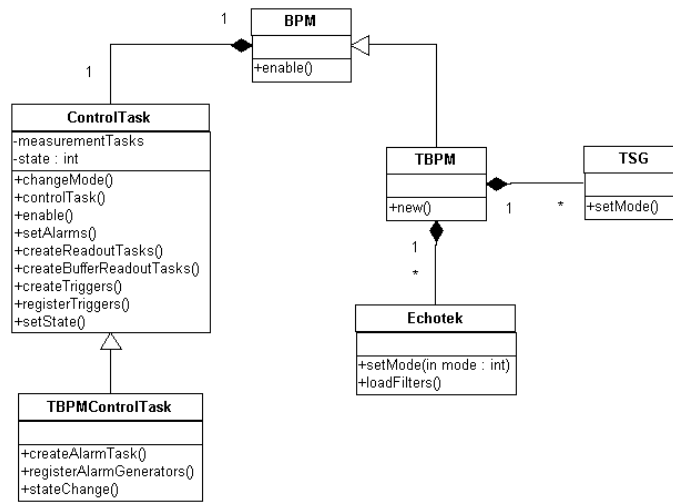

Figure 5 - Main control classes

The *BPM* class contains the entry point of the system. It is responsible for starting the *ControlTask*, which will in turn start the rest of the system.

3.2.3 Events

The system is composed of tasks and queues. The information flowing through the queues into the tasks are *events*. *Event* is the super class which has the most basic information about one event.

There can be several types of events. Those are described as subclasses of *Event*. Types of event are:

- *Alarm*: event generated by a task signaling an alarm situation;
- *Request*: generic request;
  - o *ReadoutRequest*: request to read an internal buffer;
  - o *ControlRequest*: request of some control action (e.g. change mode)
- *Reply*: generic reply for a request;
  - o *ReadoutReply*: reply for a buffer readout operation;
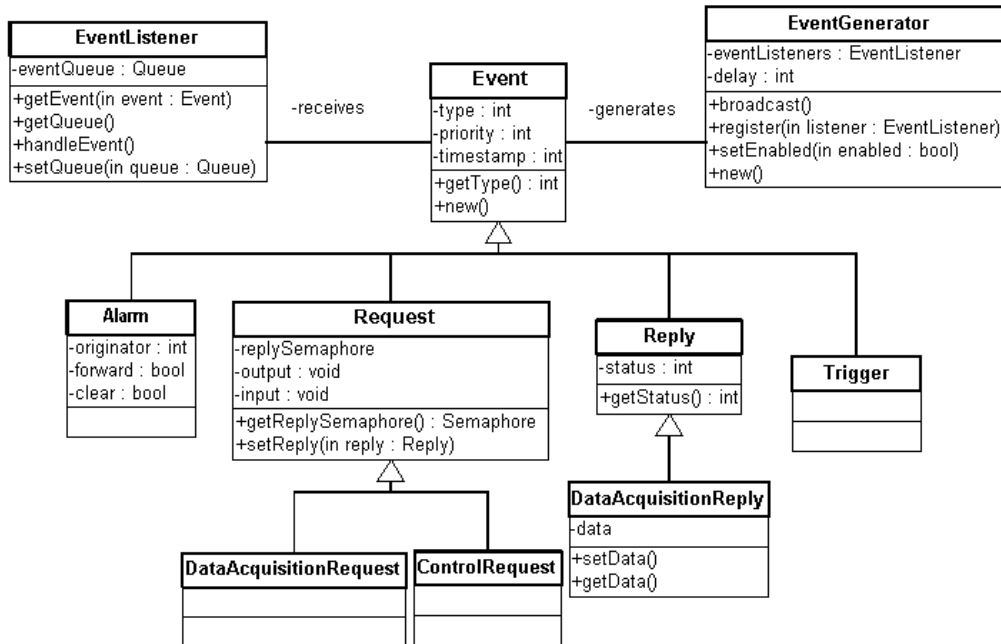
- *Trigger*: event generated on a trigger.

Figure 6 - Events in the system

An *Event* is generated by an *EventGenerator*. The *EventGenerator* has a list of *EventListeners*, to which an event is broadcasted after being generated. *EventListeners* can be dinamically added or removed from the list. The *EventListener* receives an event in its *eventQueue*. The *Event* is removed from the queue by *handleEvent ()*.

3.2.4 Event Listeners and Generators

Events can be generated and received by any entity in the system. Figure 7 shows the classes that currently deal with events. *EventListeners* are:

- *DataAcquisitionTask*: receive *Trigger* events signaling the data acquisition process;
- *BufferReadoutTask*: receive *ReadoutRequest* events when buffered data is requested.

EventGenerators are:

- *StateChangeEventGenerator*: generate a *Trigger* signaling a state device change;
- *InterruptTriggerGenerator*: generic event generator based on interrupts;
  - o *TCLKGenerator*: generate TCLK *Triggers* on interrupts;
  - o *TimeTriggerGenerator*: generate a time *Trigger* on every tick of a timer.

EventListeners **and** *EventGenerators*:

- *AlarmTask*: receives *Alarms* from other tasks in the system; and generates *Events* sent to the *ControlTask* to inform about the current alarm situation;
- *ControlTask*: receives *ControlRequests* and *Triggers*; and generates *Alarms* and *Triggers*.
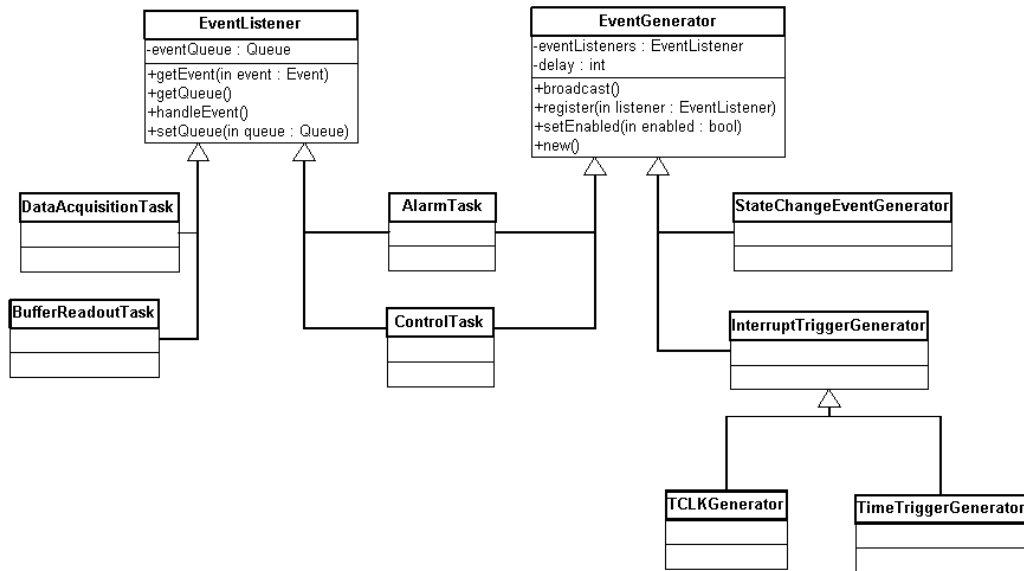

Figure 7 - Event listeners and generators (need update)

3.2.5 Data

During the data acquisition process the *DataAcquisitionTasks* perform reads from a *DataSource* (EchoTek boards) and save the result to an internal *DataBuffer*. A *DataSource* defines a generic class for reading out *DataEntries*. There can be several types of *DataSource*. For the Tevatron BPM system two of them are defined: *EchoTek* and *DataBuffer* (see Figure 8). This means that data can be retrieved either from the EchoTek boards or from an internal buffer (e.g. a task can feed the slow abort buffer with data from the fast abort buffer).

The destination of data read by the *DataAcquisitionTask* is a *DataBuffer*. It has knowledge of the *Metadata* used to tag the data, such as beam type, accelerator state and system status. All data entries are organized as *DataEntries*. The *DataEntry* can vary depending on the type of measurement.
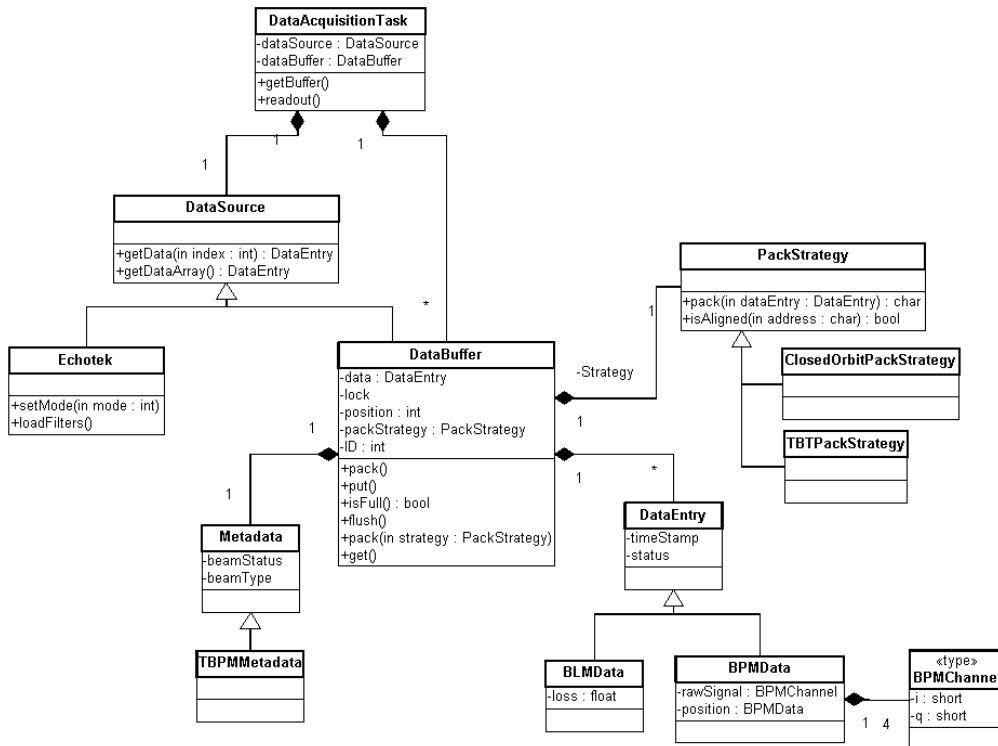
Figure 8 - Reading and saving data

The *DataBuffers* have data stored in a format that may be different from the format sent to the end user through the *BufferReadoutTasks*. Data is formated according to a *PackStrategy*. Depending on the data type and on the *ReadoutRequest* a specific *PackStrategy* is used (e.g. *ClosedOrbitPackStrategy* and *TBTPackStrategy*).

3.2.6 Alarms

The classes related handling and generating alarms are shown on Figure 9. An *Alarm* is generated by an *AlarmGenerator*. The generators in the system are the following tasks: *BufferReadoutTaks*, *DataAcquisitionTask* and *ControlTask*.

The *AlarmTask* is responsible for receiving *Alarms* generated by the *AlarmGenerators*. It declares an alarm state depending on the *Alarm* received.
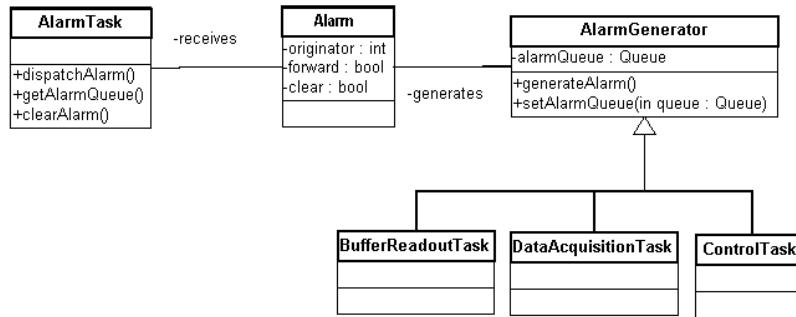
Figure 9 - Alarm classes

3.3 Activity Diagrams

This section contains diagrams showing the work flow of different tasks in the system. Figure 10 contains the basic flow for the *ControlTask*. It basically has to take care of the initialization of the system and enter a closed loop waiting for commands from its input queue. These commands are requests from MOOC, backdoor messages, alarms or triggers.

What happens if the control task gets stuck and don't read the input queue? What if a very high priority request comes into the queue? The control task has to deal with it ASAP. Perhaps there should be a pool of control tasks for reading the input queue?
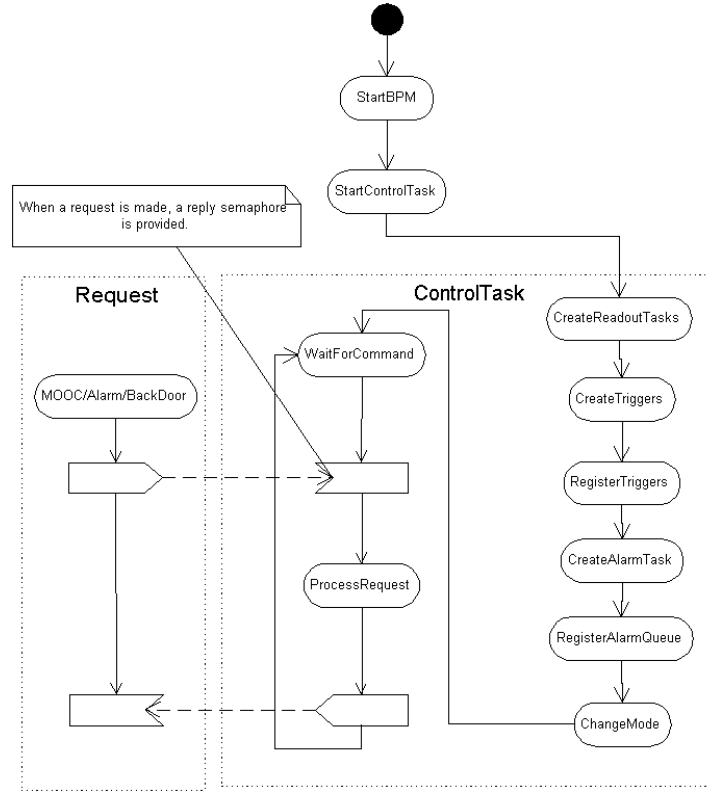

Figure 10 - *ControlTask* flow

Similarly, the *DataAcquisitionTasks* run in a closed loop waiting for *Triggers*. Upon the reception of a *Trigger*, the *DataAcquisitionTask* begins the data acquisition process from its *DataSource*, which can be a hardware or a software entity. The *DataAcquisitionTasks* are independent of each other but may share some source code (e.g. *TurnByTurn* and *InjectionTbT* in the picture).

Figure 11 depicts several *DataAcquisitionTasks*, but the functionality of some can be combined into only one task. For example, the *FastAbort* may also be responsible for the tasks performed by the *SlowAbort*. It is an implementation choice, and the final decision may be driven by the performance of the options.

The framework also allows a *DataAcquisitionTask* to generate *Triggers* to another *DataAcquisitionTask*. Suppose that there is a *InjectionTbTClosedOrbit* task. It would receive a trigger from the *InjectionTbT* task informing it that new data is in the internal buffer, and a closed orbit can be calculated.

The process of retrieving data from internal buffers is shown in Figure 12. A request comming from Mooc or backdoor is posted on the queue and one of the *BufferReadoutTasks* picks up the request, processes it and return the reply already in the format expected by the online applications (doc #860).
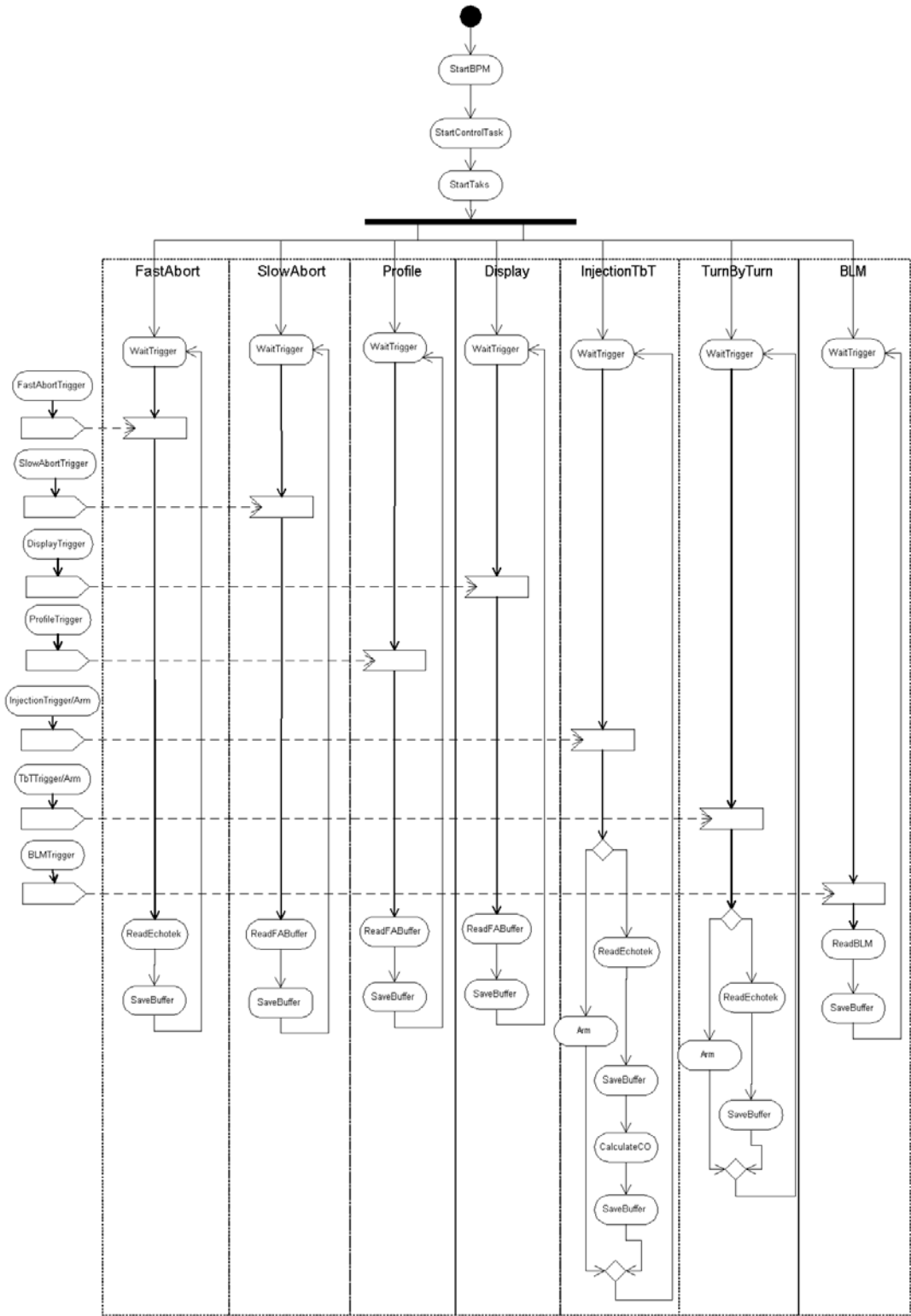
Figure 11 - *DataAcquisitionTasks* flow[3]
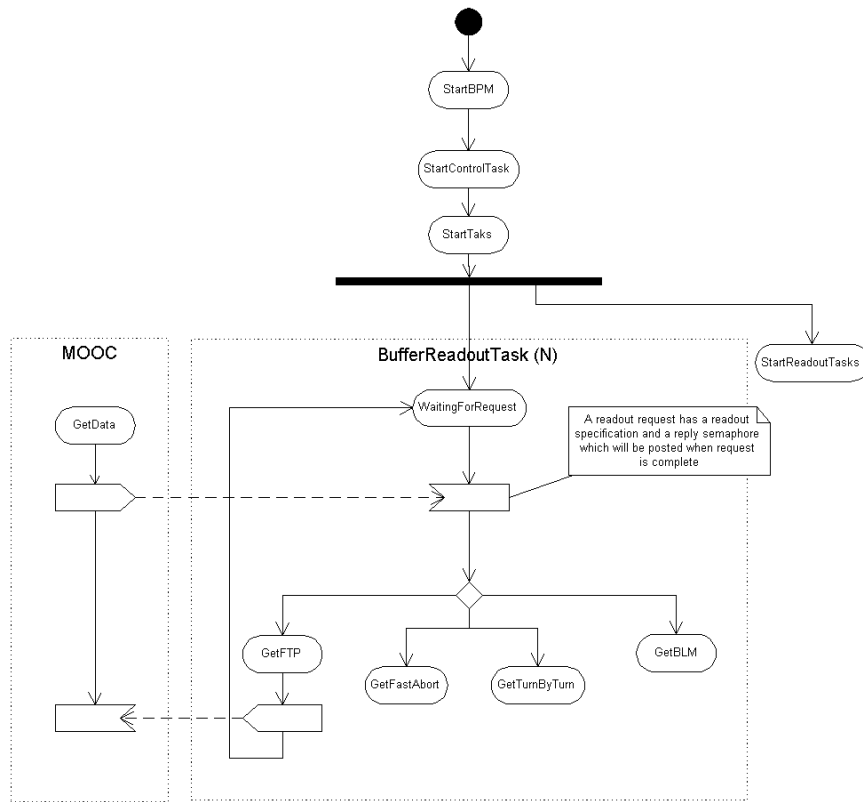
Figure 12 - *BufferReadoutTask* flow

3.4 Sequence Diagrams

This section describes common software scenarios for the front-end Tevatron software. The diagrams contain objects of the classes previously discussed and shows interactions between them throughout the course of a given scenario. The sequences shown do not correspond exactly to the implementation, but they serve as a guide to understand how objects and classes are related to each other in a dynamic environment. The flow of events starts at the top of the diagram and go downwards, following the string of method calls.

3.4.1 Initialization

Figure 13 shows how the objects in the system are first created and what are the expected operations. The entry point is the *BPM* object, which will create the *ControlTask*. The

---

[3] The tasks depicted in the picture (swimlanes) do not necessarily represent how the system will be implemented. Functionality of tasks can be combined (e.g. the InjectionTbT and TurnByTurn could be one task).

ControlTask is responsible for creating most of the objects within the system, it must instantiate the *DataAcquisitionTasks*, the *BufferReadoutTasks*, create the *AlarmTask* and create the *EventGenerators*.
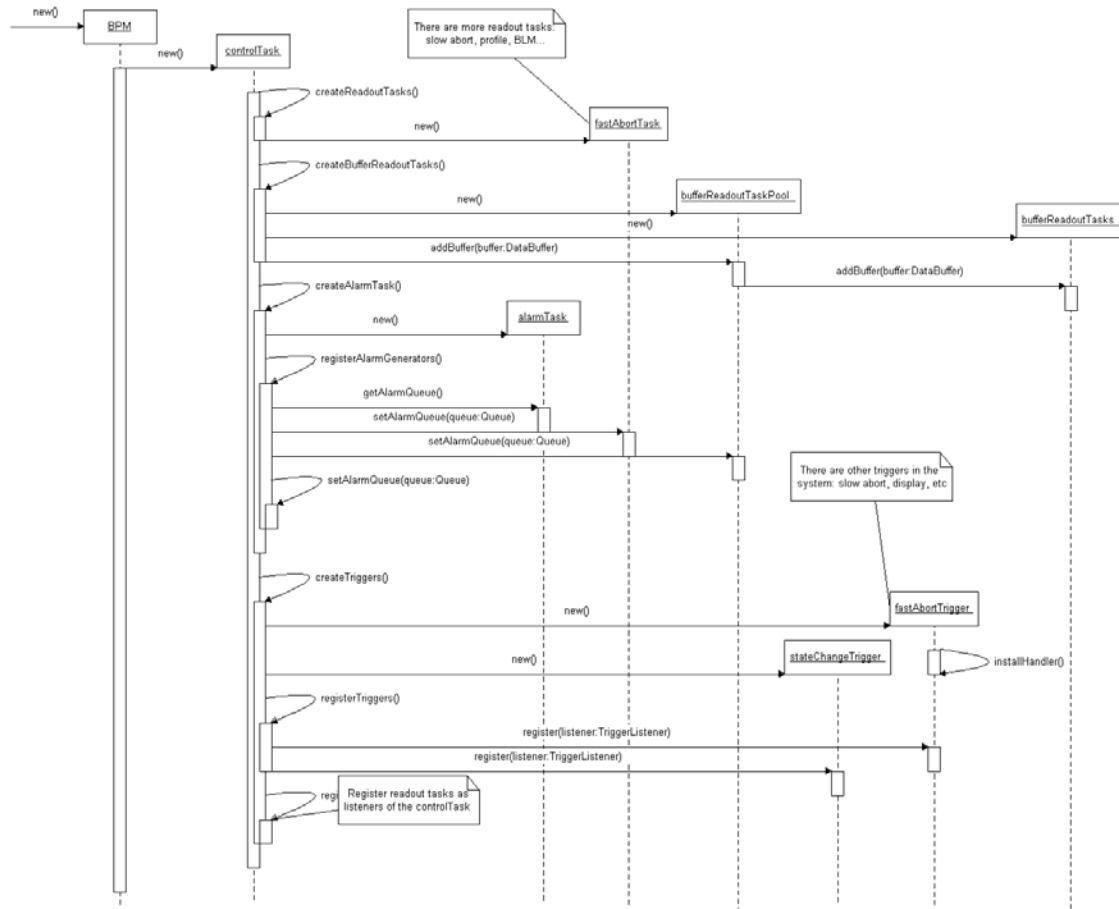
Figure 13 - Objects creation sequence

Following the instantiation of the objects in the system, the tasks need to be started in order to do the actual work of data acquisition. The *ControlTask* is responsible for getting them to work, as shown in Figure 14. After starting the tasks, the normal mode of operation is enabled through *changeMode ()*.
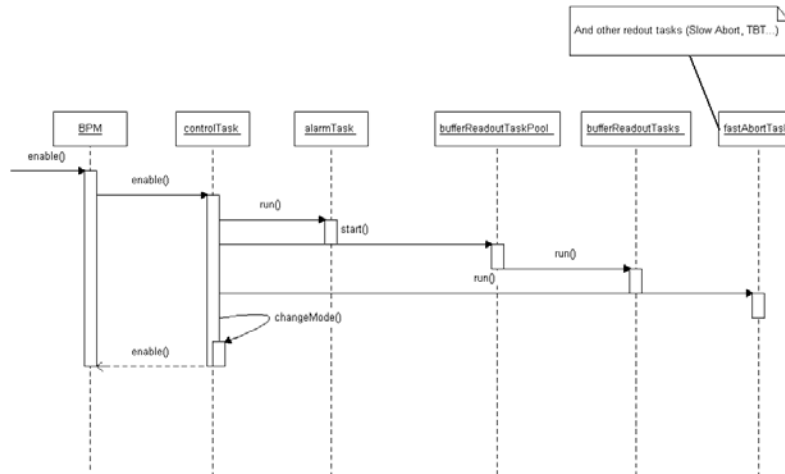
Figure 14 - Tasks initialization

3.4.2 Mode Change

The system has the ability to change modes of operation when running. The most common modes are closed orbit and turn-by-turn[4]. The closed orbit mode is the default mode of operation. Turn-by-turn mode is enabled on user request or on a programmed *Trigger*. Figure 15 shows the sequence of operations when changing from the default mode to the turn-by-turn mode.

Figure 15 - Changing modes

---

[4] There can be a turn-by-turn request when the system is already in turn-by-turn mode. In this case, the system must halt the current measurement and restart it according to the new specification

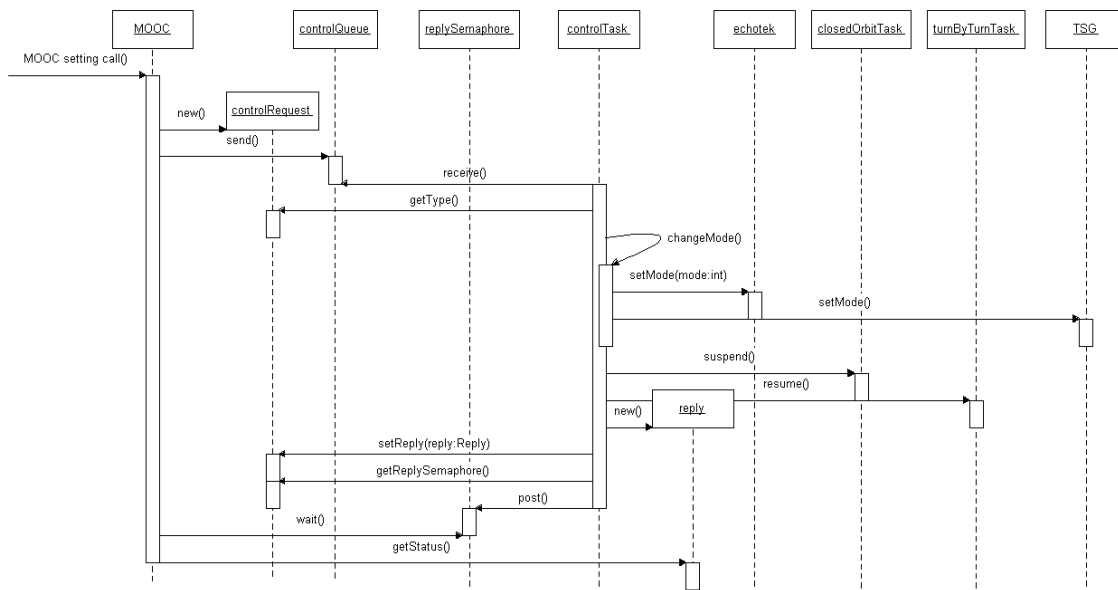In the particular case depicted on Figure 15, the mode change is triggered by an user command from MOOC. The command will be passed down to the *controlTask* as a *ControlRequest* through the *controlQueue*.

The *controlTask* is responsible for changing the mode of operation of the EchoTek boards (by loading a different configuration) and setting the timing system (*TSG*) to the turn-by-turn mode. It is also its job to suspend and resume *DataAcquisitionTasks* according to the mode of operation.

After the new mode is enabled, the *ControlTask* returns a *Reply* to the MOOC framework containing the result of the operation. In the figure the MOOC framework remains blocked in a semaphore after submitting the request and is released upon receipt of the reply, which involves a semaphore post operation.

3.4.3 Buffer Readout

Buffer readout operations follow the structure defined in Figure 16. Similarly to the situation depicted in Figure 15, a request comes through the MOOC framework, a *BufferReadoutRequest* is created and passed to the *BufferReadoutTasks*. Those have to check the type of the request, get the data from the internal buffer and pack it in the format described by the document 860.
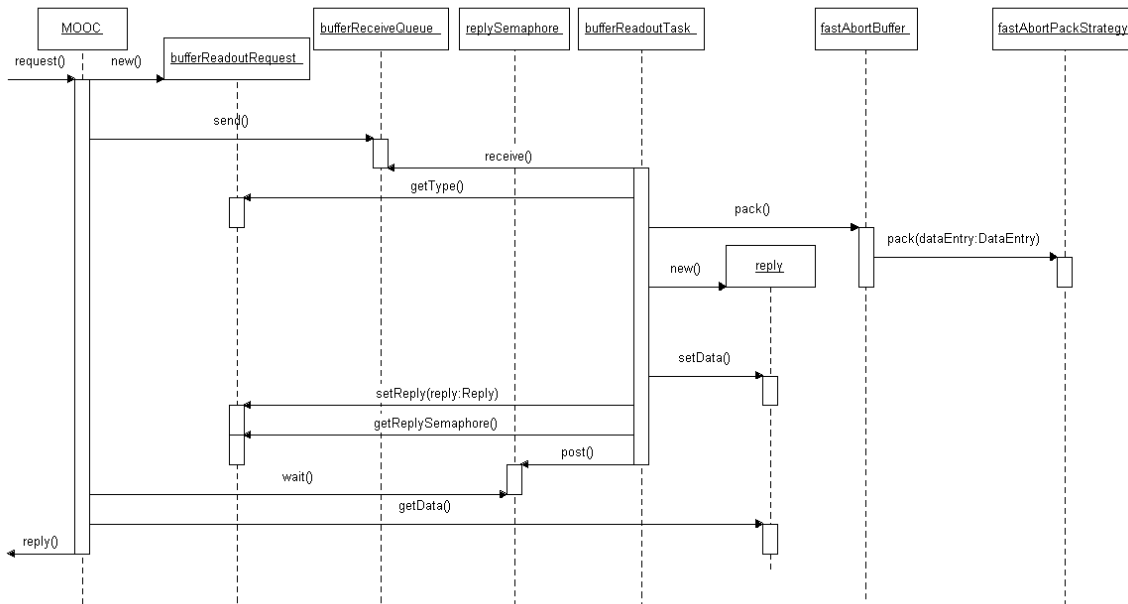

Figure 16 - Fast abort buffer readout

Data packed from the internal buffer is passed to a *Reply*, which will be received by the MOOC framework upon the completion of the operation. Similarly to the sequence

described in the above section, the MOOC framework remains blocked on a semaphore while the system is processing its request. The semaphore is released after the *Reply* is ready.

This buffer readout structure isolates MOOC from software internals. This allows standard communication with other frameworks, such as the backdoor. The price to be paid for the isolation of the code is the amount of processing involved.

For operations that require high rate of data readout, the sequence depicted in Figure 16 may not be efficient. For that purpose, it should be possible for the MOOC framework (or backdoor) to have direct access to internal buffers. Since fast time plots are nothing more than a single value, the access should be basically read the first element of the *DataBuffer*. Figure 17 shows this alternative scenario.
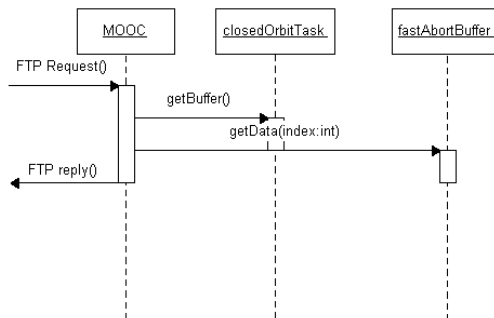
Figure 17 - Alternative fast time plot readout

3.4.4 Alarms

All tasks in the system are capable of generating *Alarms*. Figure 18 shows the sequence of an alarm generation. A task in the system creates an *Alarm* and it is sent to the *alarmQueue*, which is monitored by the *AlarmTask*. This task decides the criticality of the alarm and send out a MOOC alarm and informs the *ControlTask* that the system is in an alarm state.
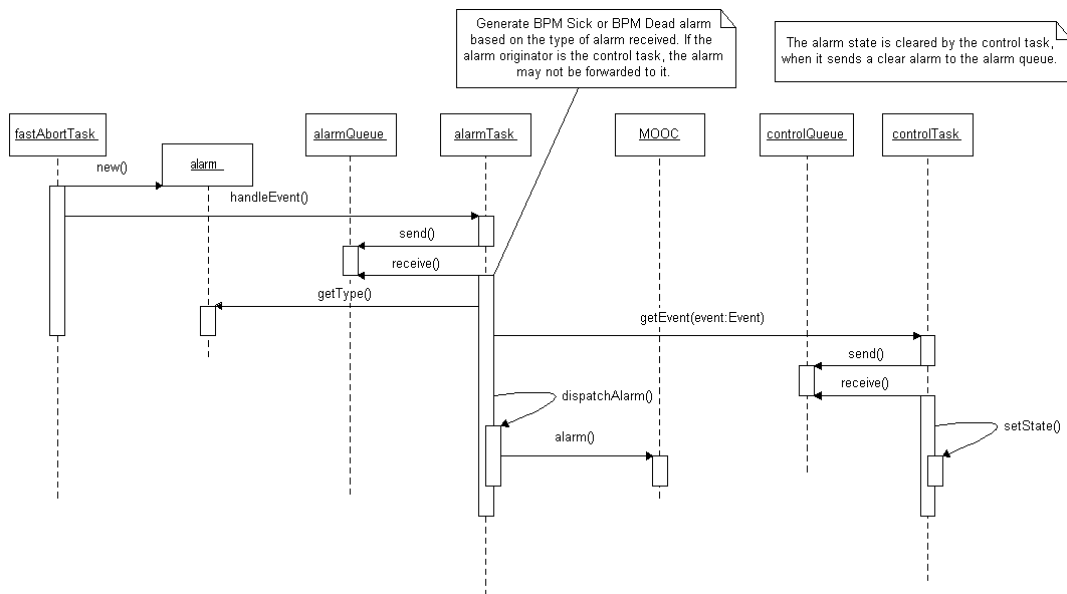
Figure 18 - Alarm generation

The alarm state can be cleared by the *ControlTask* through sending a clear message to the *AlarmTask* (see Figure 19).
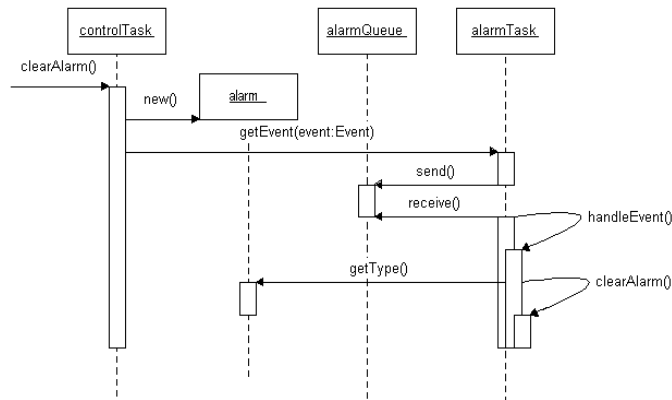
Figure 19 - Clearing an alarm

3.4.5 Events

Figure 20 shows a generic view of how an *Event* is handled in the system. The *EventGenerators* create *Events*, which are sent to *EventQueues* owned by *EventListeners*. An *EventListener* is usually a task and will receive events from its queue and process them within the *handleEvent ()* method.
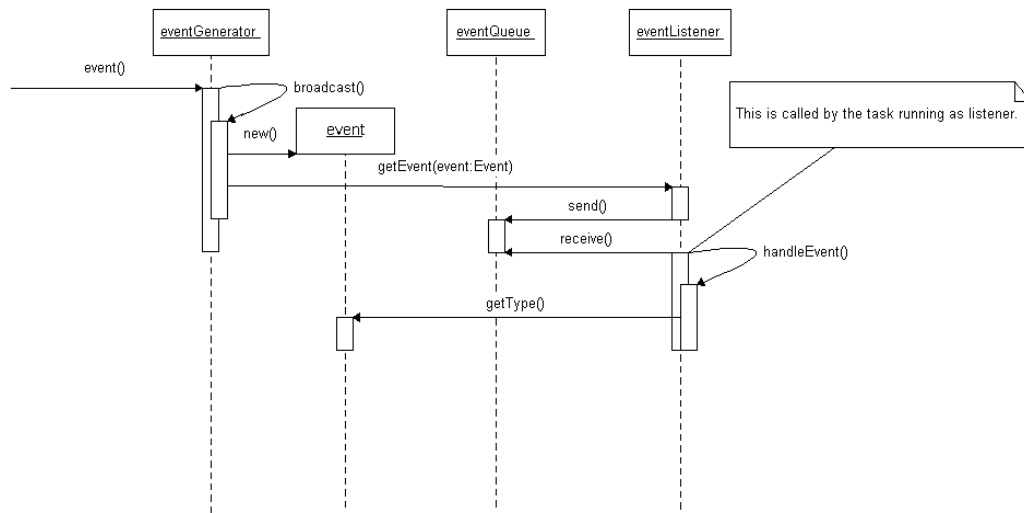
Figure 20 - Event generation

A particular case of event handling is shown in Figure 21, where the generator is of the type *StateChangeEventGenerator*, and the receiving side is the *ControlTask*.
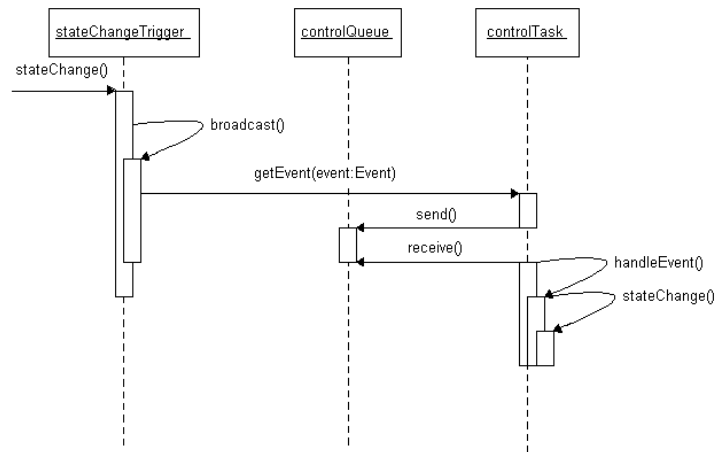
Figure 21 - State device change

3.4.6 Data Acquisition

The data acquisition process is similar to the event handling scheme on Figure 20. The event in question is a *Trigger*, generated by a *TriggerGenerator*. The event is sent to a *DataAcquisitionTask*, which in turn will acquire data from a *DataSource* and save it to a *DataBuffer*. This process is illustrated in Figure 22.
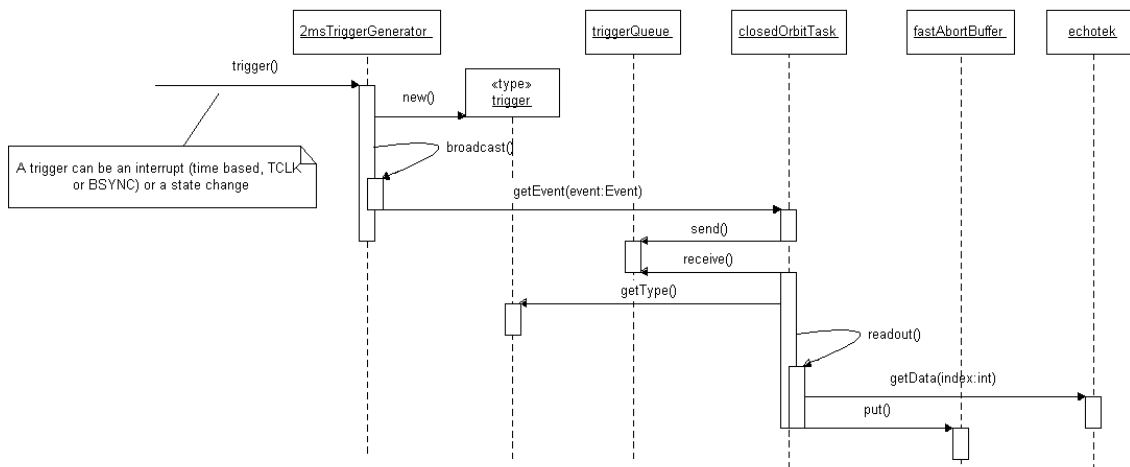
Figure 22 - Fast abort trigger generation

3.5 Packages

The system is designed to provide a flexible and generic framework for BPM projects for other machines besides the Tevatron. For this purpose, the classes are divided in a generic BPM class package and Tevatron BPM package.

3.5.1 Generic BPM classes

These are generic classes that form the BPM framework. These should be able to provide a running system with hooks for machine specific code. It contains classes that provide data acquisition, control, configuration management, alarms and buffering.

- BPM
- Task
    - ControlTask
    - BufferReadoutTask
    - DataAcquisitionTask
    - AlarmTask
- Queue
- Semaphore
- EventGenerator
    - StateChangeEventGenerator
    - InterruptTriggerGenerator
    - TCLKGenerator
    - TimeTriggerGenerator
- AlarmGenerator

- Event
  - Alarm
  - Request
    - ReadoutRequest
    - ControlRequest
  - Reply
    - ReadoutReply
  - Trigger
- DataSource
  - DataBuffer
- Metadata
- DataEntry
- PackStrategy
- TaskPool
  - BufferReadoutTaskPool
- CalibrationSystem
- TimingSystem

3.5.2 Tevatron BPM classes

These are the classes that implement the specific BPM behavior for the Tevatron machine. Code for specific hardware and Tevatron alarms must be implemented in these classes.

Superclass	Subclasses
BPM	*TBPM*
DataAcquisitionTask	*TBPMFastAbortTask*
	TBPMSlowAbortTask
	TBPMDisplayTask
	TBPMTurnByTurnTask
	TBLMFastAbortTask
	TBLMDisplayTask
AlarmTask	*TBPMAlarmTask*
DataSource	*EchoTek*
TimingSystem	*TSG*
Metadata	*TBPMMetadata*
PackStrategy	*TBPMClosedOrbitPackStrategy*
	TBPMTurnByTurnPackStrategy
	TBPMTimeSlicedPackStrategy
	TBLMPackStrategy
	TBLMTimeSlicedPackStrategy
DataEntry	*TBPMData*
	TBLMData
	TBPMChannel
EventGenerator	*TCLKEventGenerator*

StateChangeEventGenerator

3.5.2.1 TBPM Buffers

Even though the data source of position, intensition and loss data are only the EchoTek boards and the BLM chassis, the system must keep several types of data in different buffers. Some of these data may be the same, what makes them different is the event that triggered its acquisition. According to the AD document #903, these are the buffers that must be implemented in the system:

BPM buffers:
- Fast Abort Buffer (array)
- Slow Abort Buffer (array)
- Fast Time Plot Buffer
- Profile Frame Buffer (array)
- Display Frame Buffer
- Snapshot Buffer

BLM buffers:
- Fast Abort Buffer (array)
- Display Buffer
- Fast Time Plot Buffer

These buffers in the system are realized by the class *DataBuffer*. The Fast Time Plot buffers are the only ones that do not have its own memory location. When handling a FTP request the system will return the latest value from the Fast Abort Buffer.

Data from the buffers are read by the *BufferReadoutTask* and organized by a *PackStrategy* according to the structures defined in the AD document #860.

3.5.3 Implementation

This section is a guideline for the implementation of the system. It is divided into two parts: the first one contains the elements related to the generic BPM framework and the second lists the components of the Tevatron system.

It is highly recommended that every class have a unit test associated to it. The tests should call all methods from the classes and check the returning data and status.

3.5.3.1 Building the generic framework

Implementation of classes is independent, otherwise noted that there are requirements. First level of elements can be implemented in parallel, while elements within (a through z) usually require sequential implementation.

1. Buffers
   a. Implement *DataSource*
   b. Implement subclass that generates a known pattern (e.g. *TestDataSource*)
   c. Implement *DataEntry*
   d. Implement Subclass of *DataEntry* (e.g. *TestDataEntry*)
   e. Implement *PackStrategy*
   f. Implement subclass of *PackStrategy* (e.g. *TestPackStrategy*)
2. Wrappers
   a. Implement *Task*
   b. Implement *Queue*
3. Events (requires 2b)
   a. Implement *Event*
   b. Implement *EventListener* (requires 2b)
   c. Implement *EventGenerator*
   d. Implement subclasses of *Event* (*Trigger*, *Request*, *ReadoutRequest*, *ControlRequest*, *Reply*, *ReadoutReply*)
   e. Implement *InterruptEventGenerator*
   f. Implement *TimetEventGenerator*
4. Alarms
   a. Implement *Alarm* (requires 3a)
   b. Implement *AlarmGenerator*
5. Tasks (requires on 2a)
   a. Implement *ControlTask*
      i. Implement *Metadata*
   b. Implement *DataAcquisitionTask*
   c. Implement *BufferReadoutTask*
   d. Implement *AlarmTask*
6. Task Pool
   a. Implement *TaskPool* (requires 2a)
   b. Implement *BufferReadoutTaskPool* (requires 5p)

At the end of the implementation of the generic framework it is expected to have a test version of the system running, generating fake data, receiving comands and requests from users.

3.5.3.2 Building Tevatron BPM

The implementation of all classes for the Tevatron specific system are independent. Any requirement on input/output data can be fulfilled by using Test classes from the generic framework. For example, it is not necessary to have the EchoTek class in place to generate data, one can use the *TestDataSource* class for that purpose or implement a *TestEchoTek* class which generates simulated data.

- BPM hardware
  a. Implement *EchoTek*
- BLM hardware
  a. Implement *TBLM*
- Timing system
  a. Implement *TSG*
- Control
  a. Implement *TBPMControlTask*
- Data
  a. Implement *TBLMData*
  b. Implement *TBPMData*
  c. Implement *TBPMChannel*
  d. Implement *TBPMClosedOrbitPackStrategy*
  e. Implement *TBPMTurnByTurnPackStrategy*
  f. Implement *TBPMTimeSlicedPackStrategy*
  g. Implement *TBLMPackStrategy*
  h. Implement *TBLMTimeSlicedPackStrategy*
- Data acquisition
  a. Implement *TBPMFastAbortTask*
  b. Implement *TBPMSlowAbortTask*
  c. Implement *TBPMDisplayTask*
  d. Implement *TBPMTurnByTurnTask*
  e. Implement *TBPMInjectionTurnByTurnTask*
  f. Implement *TBPMProfileTask*
  g. Implement *TBLMFastAbortTask*
  h. Implement *TBLMDisplayTask*
- Events
  a. Implement *StateChangeEventGenerator*
  b. Implement *TCLKEventGenerator*
- Alarms
  a. Implement *TBPMAlarmTask*

4 Appendix

4.1 The Recycler Software

The current recycler software provides configuration, read out and diagnostics capability for a hardware setup specific to get proton position in the recycler ring. The hardware involved consists of:

- Motorola MVME processor board with two IP320 ADC and one IPUCD TCLK/MDAT
- IP carrier containing four IPTSG timing signal generator modules developed at Fermilab.
- EchoTek boards

The software configures all timing devices and the echotek boards according to read out specifications passed through ACNET. The software allows the recycler BPM to run in the following modes:

- Background Flash;
- Flash;
- Closed Orbit;
- Turn-by-Turn;
- Turn-by-Turn Scan.

The following subsections describe the current software from the viewpoint of management, control and buffering techniques.

4.1.1 Control

The system has a main control task (ArmEventTask), which is responsible for spawning tasks for handling specific modes of operation. For example: RepetitiveFlashTask and TurnByTurnTask.

When switching modes, the ArmEventTask has to start a new task that has the ability to configure and conduct the read out for requested mode. When starting the new mode the new task is responsible for configuring the hardware (EchoTek board and timing system).

The communication between tasks is handled through message queues, for example to signal events such as measurement complete. Message queues are also used to signal triggers received by interrupt handlers.

4.1.2 Buffering

The data read out by the tasks is kept in buffers visible inside the BPM class (which is the main class in the system). The buffers have predefined size and are managed by the task

that currently use it. Since there is only one readout task running at a time, there are no writing conflicts. Data can be read from the buffers via ACNET or backdoor requests.

4.1.3 Readout

Readout tasks wait for signals from the message queue. The interrupt handlers receive the external trigger and send data to the queue, which is immediately received by the readout task.

4.1.4 ACNET Communication

In parallel to the readout, the software can receive ACNET/MOOC calls to retrieve data from the buffers. The software is accessed through callbacks, which extract raw data from the BPM buffers and calculates the positions and intensities. The data is packed and returned in the response to the online software.

4.1.5 Debug and Diagnostics

The page R33 provides ways to configure the recycler BPMs, such as selection mode of operation, enabling/disabling diagnostics and setting timing constants. It is possible to control any recycler crate from the software. The communication between R33 and the recycler software is via ACNET.
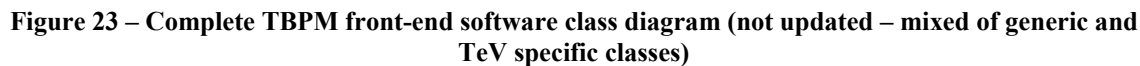
The recycler software also uses the backdoor scheme for getting data using and alternative way to ACNET. It is a client/server facility, where the server is the front-end and the client is a remote application, currently labview.

The server receives requests to change configuration, return data, turn diagnostics on among others. The server is programmed by adding *Accessors*, which will serve as bridge to the internals of the system.

A new version of the backdoor software allowing peer to peer communication is planned.

4.1.6 Alarms

Not implemented in the current version of the recycler software.

4.2 Class Diagram

Figure 23 – Complete TBPM front-end software class diagram (not updated – mixed of generic and TeV specific classes)

5 Bibliography

[Cockburn] Alistair Cockburn, *Writing Effective Use Cases*, Addison-Wesley, 2001.

[Fowler] Martin Fowler, *UML Distilled: A brief guide to the standard object modeling language*, Addison-Wesley, 2003.